NASA Contractor Report 191445
ICASE Report No. 93-14

ICASE

20 Years of Excellence

RELIABILITY ANALYSIS OF COMPLEX MODELS USING SURE BOUNDS

David M. Nicol
Daniel L. Palumbo

NASA Contract Nos. NAS1-18605 and NAS1-19480
March 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

Reliability Analysis of Complex Models using SURE Bounds

David M. Nicol[1]
College of William and Mary
Williamsburg, VA 23187
and
Daniel L. Palumbo
NASA Langley Research Center
Hampton, VA 23668

ABSTRACT

As computer and communications systems become more complex it becomes increasingly more difficult to analyze their hardware reliability, because simple models may fail to adequately capture subtle but important model features. This paper describes a number of ways we have addressed this problem for analyses based upon White's SURE theorem. We point out how reliability analysis based on SURE mathematics can be extracted from a general C language description of the model behavior, how it can attack very large problems by accepting recomputation in order to reduce memory usage, how such analysis can be parallelized both on multiprocessors and on networks of ordinary workstations, and observe excellent performance gains by doing so. We also discuss how the SURE theorem supports efficient Monte Carlo based estimation of reliability, and show the advantages of the method.

[1]This research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18605 and NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681. This research was also supported in part by NASA grants NAG-1-1060, NAG-1-1132, and NSF grant CCR-9201195.

1 Introduction

White's SURE theorem has laid the foundation for a number of reliability tools, including SURE [3] itself, ASSIST [2, 11], TOTAL, and PAWS. The latter three tools provide the user with a formal framework within which a model is described, then use the model description to explicitly build a semi-Markov state-space. The tool SURE is then applied, determining upper and lower bounds on the transient probability of the system entering a state reflecting system failure (i.e., a *death-state*) within a specified period of time. These tools have a large user base and have proven to be very useful in a wide range of contexts. For example, a survey conducted by NASA Langley found that the ASSIST/SURE toolset is used by United Technologies to model redundant engine control architectures, by Boeing to model fighter flight control systems, by Raytheon to model space-borne systems, by Rockwell-Collins to evaluate trade-offs in the reliability and safety of primary flight control architectures, and General Electric to model engines, engine controllers, locomotive engines, and satellite controllers. Industrial interest in SURE-based analysis is apparently strong.

One drawback of these tools is that they are unable to efficiently explore (i) very large models, (ii) models where the state transformation cannot be expressed in terms of simple modifications to state variables, or (iii) models where recognition of a death-state is complex. For example, model sizes become large any time one desires a complete analysis of a detailed model; state transformations become complex if recovery transitions involve non-trivial computations, such as finding new routes for messages through a fault-tolerant network; death-state recognition may be complex if system operability is defined in terms of the system's ability to provide some service, e.g., every pair of operable processors are able to communicate using some specific routing protocol. We later give examples of all three situations.

This paper describes methods we have used to address these situations, and a software tool called ASSURE that embodies these methods. ASSURE combines the functions of ASSIST and SURE. The user's interface to ASSURE is an enhanced version of the ASSIST [2] language. ASSIST's power of expression is extended to almost arbitrarily complex models by allowing the user to write C language routines to recognize system failure, to recognize system transition conditions, and to express system state modification following a transition. Other techniques we describe are related to using the SURE bounds to efficiently analyze some large models. One such method is to concurrently generate and analyze a model's state-space via depth-first-search (DFS) exploration. Memory requirements are limited to that needed to manage the DFS stack, instead of the entire state space (as is presently required with ASSIST/SURE); however, memory efficiency comes at the price of state recomputation. The method has the intentional and important advantage of supporting parallel processing on ordinary networks of workstations. We also investigate user-assisted methods for trimming the model space. Here we permit the user to use expert knowledge of system behavior to classify possible state transitions, thereby allowing the tool to make inferences about the possibility of imminent system failure without actually generating the model states that

might reflect such failure. Even with the fore-mentioned features, the sheer size of state-spaces involved in some models prohibit an exact and exhaustive analysis. To address this problem we have developed efficient ways of jointly using Monte Carlo simulation and the SURE bounds to construct confidence intervals on estimated upper and lower reliability bounds. In addition, our method supports estimation of arbitrary measures of system performance in death-states, and we have extended the ASSIST language to support automated estimation of these user defined statistics. Finally, the Monte Carlo analysis is easily parallelized as well, again on a network of ordinary workstations.

ASSIST/SURE is only one of many good reliability tools; i.e., see the recent survey [8]. Various of the features we've incorporated into ASSURE have been used in the past by other tools. The notion of expressing models in a high level language and then automating the generation and analysis of the underlying Markov chain is common to all modern reliability tools. For example, HARP [5] uses a fault tree description of failure processes and a petri-net description of recovery processes. From these a Markov chain is constructed and analyzed to provide system state probabilities. SAVE [7] uses a language describing a machine-shop with repairmen. SHARPE [17] provides a number of different model types, in a sort of analysis toolbox. The notion of truncating a state-space (while developing it, or searching it) is found in the tools above, as well as in [6]. The idea of using a common programming language as a vehicle for describing a model is exploited in DEPEND[9], which also uses Monte Carlo simulation, as does SAVE [7]. A Monte Carlo version of HARP has also been developed [1]. Our intent is to show how ASSURE's features together allow us to attack very large and complicated system models, and to demonstrate a single tool that seamlessly allows either an exact analysis or a simulation analysis and, or, a serial solution or a parallel solution from a common (but general) model description. Our main contributions are implementation methods suitable for solving such models. These contributions are three-fold. First, we demonstrate that on an interesting set of large problems there is much to be gained by regenerating states in a depth-first analysis, rather than saving each generated state against the possibility that it will be visited again. This style of analysis permits solution of some models considered to be "out of reach" at the time [8] was written (i.e., 10^5 states, 10^{10} ratio of repair rates to component failure rates). One should note, however, that the relative advantage of the method decreases as the number of failures required to push the system into a death-state increases. Consequently, exact analysis using the method is best suited for systems that tolerate 2-5 failures in the mission time. We demonstrate empirically that this approach is ideal for parallel processing—a new and highly practical aspect of reliability analysis. Thirdly, we show how the SURE bounds lend themselves to an efficient Monte Carlo analysis, which itself is parallelizable.

It might be argued that detailed analysis of large models is unnecessary, since at some level a reliability model will have to mask details anyway, and an expert modeler can often craft a good model from a detailed understanding of the system being modeled. While we will never dispute the power of a expert modeler using a simple tool, we believe that the need to analyze large

detailed models is inevitable. We anticipate the day when a system is specified and designed using a single tool from which reliability and performance analyses are automated. An automatically generated model is far more likely to be large and complicated than one developed by a human expert. Furthermore, an automated analysis accommodates a system design or parameter change by simply redoing the analysis—that same change may invalidate a human expert's entire approach. Towards this end, we are exploring ways in which large complex models might be automatically analyzed.

It might also be argued the SURE approach is inadequate, owing to its assumption of time-independent failure rates. While this argument has some validity, we are not attempting to advance any particular side in the sometimes heated debate over reliability tools. We believe that the techniques we describe are not limited to SURE; they can be applied to any mathematical analysis based on paths through a state-space. Perhaps the potential shown by ASSURE for large problems may motivate mathematical research on path-based analysis that overcomes SURE's limitations.

This paper is organized as follows. Section 2 describes two model problems that exhibit challenging characteristics. Section 3 describes extensions we've provided for the ASSIST language to enhance model expression. Section 4 presents the SURE bounds. Section 5 describes implementation techniques that support the analysis of large complex models, and Section 6 explains how SURE bounds can be used in the context of an efficient Monte Carlo analysis. Section 7 presents our conclusions.

2 Two Examples

Our work has been motivated in large part by the challenges presented by two diverse yet representative reliability models. The first model is of a fault-tolerant flight-control computer network having a complex recovery mechanism, the second is that of a large computer network that achieves fault-tolerance through redundancy of communication channels. This section discusses both models, and the characteristics which challenge the capabilities of existing SURE-based tools.

The first problem presents the challenge of state-space size, and complexity of expressing a complex reconfiguration strategy within the confines of the modeling language. These challenges are both present in a model based roughly on AIPS [12], an architecture developed by Stark Draper Labs. The model is comprised of a Fault-Tolerant-Processor (FTP), that manages a collection of "devices" (sensors). The devices are replicated four times for quad redundancy, and are distributed across two networks, accessed by the FTP from six channels. Only selected links in the network are "in use" at any time. The set of selected links in a network establish a virtual bus between one FTP channel, and every operational node in the network. In the event a selected link or a network node fails, the network is considered to be down. However, it may be repaired if another set of links can be found to establish the virtual bus. During recovery the FTP knows to ignore the downed network, and to take its sensor data from the other network. The system is considered

to have failed if the FTP itself fails, if both networks are simultaneously down, or if the majority of operable devices of any type are not able to communicate with the FTP. Figure 1 illustrates an example of this network and its hardware components. Shown are four channels linking the FTP to the networks, six network interfaces, thirty-two links, fourteen switching nodes, eight interface devices, and sixteen devices. "In-use" links in one particular system state are highlighted; a number of links are shown to have failed.

The particular set of links chosen during repair to re-implement a virtual bus will impact the distribution of the remaining time until system failure, especially if failure rates of remaining components are heterogeneous. Greater accuracy is obtained then by explicitly modeling the reconfiguration process than by assigning an approximate recovery rate to a network failure. If we accept the desirability of an accurate recovery model, we consequently require that the reliability tool be able to concisely express the reconfiguration strategy.

The second problem arose in a study comparing the effectiveness of fault-tolerant routing protocols on a binary hypercube. Nodes and links may fail; when one does, no explicit recovery is attempted. However, network messages can accommodate such failures by adaptively rerouting around failed nodes or links. A variety of fault tolerant routing protocols exist, some of which may not find a extant route. Given a protocol, the system is considered to have entered a death-state if either more than half of the nodes have failed, or if there exist two operable nodes between which the protocol cannot establish a message path. The complexities of these protocols defeat more elegant graph-based based reliability analyses, and we are left to use simulation if we are to estimate reliability.

This model presents us with two fundamental problems. First, depending on the network size, tens to hundreds of link and node failures can be tolerated before the system enters a death-state. The size of the state space absolutely prohibits an exhaustive analysis. The second problem is that recognition of a death-state is expensive. Given the state of the network, one must essentially simulate message routing behavior between every pair of nodes. The cost of a single connectivity check is $O(L)$, implying an $O(LN^2)$ death-state recognition cost.

The sections to follow describe the methods we've used to address the challenges posed by these problems.

3 Language Extensions to ASSIST

The ASSIST language (see [2]) provides a simple means of describing a system and how it evolves in the presence of failures and recoveries. The notion of state variable is central to ASSIST; one of the first roles of an ASSIST model is to declare the state variables (and their initial values) just as variables are declared in programming languages. Evolution of the system is described in terms of Boolean conditionals on the state variables (describing conditions under which a transformation may occur), and simple modification of state variables (describing the transformation itself). For

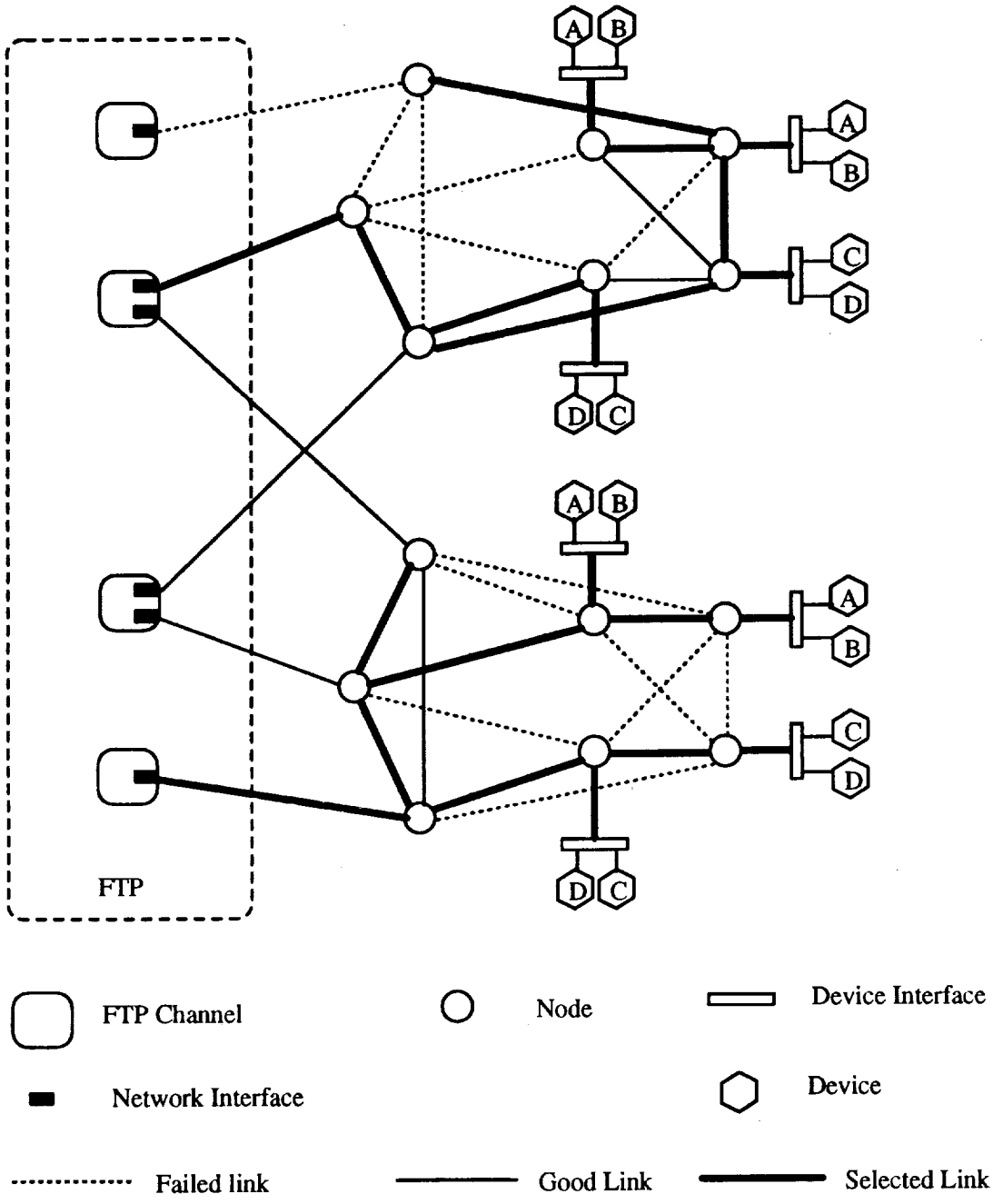

Figure 1: Example of reconfigurable flight control computer network, highlighting virtual bus connections.

example, a state variable N may describe the number of working processors, any of which may fail. The ASSIST statement

```
IF N>0  TRANTO  N=N-1  BY N*LAMBDA;
```

declares that from any system state where N exceeds zero, another processor can fail, and change the system state by decrementing N. The mathematics of SURE assume that a component's lifetime is exponentially distributed; the statement above declares that the transformation occurs with rate N*LAMBDA (LAMBDA is defined as a constant elsewhere). Boolean conditionals also identify death-states, for instance,

```
DEATHIF N=0;
```

declares that the system is in a death-state whenever all processors have failed.

This particular example is unrealistically simple. Larger ASSIST models employ compound Boolean expressions as conditionals, and modify several state variables as a result. For instance, the statements below were taken from a working ASSIST model.

```
(* COVERAGE *)
 DEATHIF (FT[1]+FT[2]+FT[3]+FT[4] ) <=
        (FF[1]+FF[2]+FF[3]+FF[4] );

(* EXHAUSTION *)
DEATHIF FT[1] + FT[2] + FT[3] + FT[4] < 2;
DEATHIF NI[1] + NI[2] + NI[3] + NI[4] + NI[5] + NI[6] < 1;

(*  TRANSITION RULES  *)
FOR I=1,6;
  IF NO1G[I]=1 TRANTO NO1G[I]=0, P[1]=0, CT=CT+1 BY LNO;
  IF NO2G[I]=1 TRANTO NO2G[I]=0, P[2]=0, CT=CT+1 BY LNO;
ENDFOR;
```

Here we see that ASSIST allows arrays of state variables, multiple DEATHIF and TRANTO statements, and looping constructs. An important aspect of ASSIST models is that they are essentially algorithmic. The TRANTO statements give a set of rules; any time the system state satisfies a rule, a transition from that state is possible. The statements following the keyword TRANTO describe how the system state is correspondingly modified, and the statement following keyword BY gives the transition rate.

We found that the ASSIST syntax for describing state modification was too limited to efficiently express the dynamic network reconfiguration required by our first model problem. There, given the operational status of network links, nodes, and devices, we must apply an algorithm to find a subset of these components that form a bus. Nevertheless, we saw that it was still possible to exploit the essential idea behind ASSIST, which is to express state transitions in terms of recognizing when and how they occur. Our simple extension is to allow the statement following a TRANTO to be a call to a subroutine in the C programming language, where declared ASSIST state variables may be

both read and written directly. Similarly, the ability to express DEATHIF and TRANTO conditions is extended by allowing calls to C routines that analyze the variables of the present system state and return a Boolean value indicating whether a particular condition is satisfied. To our knowledge, ASSURE is the only tool which both provides an analytic solution (as opposed to only simulation), and allows manipulation of model state variables by a general programming language. In our experience this ability proved invaluable when describing complex reconfiguration strategies, and when analyzing models with complex death-state conditions. In support of these extensions, we also allow a user to write a subroutine to compute the initial system state variable values, and to build static C data structures (which ought only to be read, not modified) for use by other routines. For example, we've used this feature to describe static network topologies and let the system state vector contain only the operational status of each component.

These extensions are conceptually simple, and are implemented by using an ASSURE-to-C source code translator. The translator parses the ASSIST model, translates references to ASSIST state variables into references to C variables, and uses the ASSIST model structure to create problem-dependent C subroutines for detecting death-states and for generating all transitions possible from a given state. These subroutines are compiled and linked to pre-compiled problem-independent code that controls the generation process and performs the SURE analysis. On most models, the translation step requires a few seconds and the compilation/linking step requires a few tens of seconds, on ordinary workstations. This relatively small front-end cost is easily amortized when a large model's execution phase takes minutes, or longer.

4 The SURE Theorem

Subsequent discussions are better understood following a brief description of the SURE theorem. A fuller treatment of these bounds are given in [3].

We may think of a semi-Markov state-space as a directed graph whose nodes represent states, and whose edges represent transitions. A precise mathematical definition can be found in many standard texts, e.g., [16]. The SURE theorem applies to semi-Markov processes with two types of transitions. *Slow* transitions are exponentially distributed, with small transition rates as compared with the *fast* transitions, that may have general distributions. Slow transitions typically model hardware component failure, whereas fast transitions model repair processes. The difference in transition rates may span several orders of magnitude.

The sequence of transitions defining a path through a semi-Markov state space reflect a possible system behavior in time. The amount of time the system takes to traverse a given path p is random, call it S_p. Given a *mission time* T, the SURE theorem gives formulae for upper and lower bounds ($U_p(T)$ and $L_p(T)$, respectively) on $\Pr\{S_p \leq T$, path p is taken$\}$. These bounds are of particular interest when the last state on p is a death-state.

Let $\mathcal{D}$ be the set of death-states, let I be the initial system state, and let $\mathcal{P}$ be the set of all

paths from I through states not in $\mathcal{D}$, to some member of $\mathcal{D}$. The probability that the semi-Markov process enters $\mathcal{D}$ within time T is

$$\Pr\{\text{Death state entered within time } T\} = \sum_{p \in \mathcal{P}} \Pr\{S_p \leq T, \text{ path } p \text{ is taken}\}. \tag{1}$$

To use the SURE bounds one discovers and analyzes every path in $\mathcal{P}$ (at least the ones with sufficient probability) as follows. We classify every state on a path p as being a class 1,2, or 3 state. A state is in class 1 if its transition on p is slow, and every other transition from the state is also slow. Any state whose transition on p is fast is in class 2; the transition from a class 3 state is slow, and there is at least one fast transition from that state. The following class-specific parameters are needed to state the SURE bounds.

Class 1 Let k be the total number of class 1 states on p. For the i^{th} class 1 state define λ_i to be the rate of the transition out of the state, and define γ_i to be the sum of rates of all other transitions from that state.

Class 2 Let m be the total number of class 2 states on p. For the i^{th} class 2 state define ϵ_i to be the sum of rates of all slow transitions from it. Let ρ_i be the probability that the particular transition on p is successful (as opposed to some other transition from that state); let $\mu_{2,i}$ and $\sigma_{2,i}$ respectively be the conditional mean and standard deviation of the state holding time, given that the selected transition on p is successful.

Class 3 Let n be the total number of class 3 states on p. Let α_i be the rate of the transition out of the i^{th} class 3 state on p, and β_i be the sum of rates of all other slow transitions from that same state. Define $\mu_{3,i}$ and $\sigma_{3,i}$ to be the mean and standard deviation of the holding time in that state, given that a fast transition occurs (instead of the slow transition that did occur).

Finally, let $Q(T)$ be the probability of traversing by T a path constructed by concatenating the k class 1 states, and let $r_1, r_2, \ldots, r_m$, and $s_1, s_2, \ldots, s_n$ be strictly positive numbers such that $T > \Delta = r_1 + r_2 + \cdots + r_m + s_1 + s_2 + \cdots + s_n$. Then

$$L_p(T) \leq \Pr\{S_p \leq T, \text{ path } p \text{ is taken}\} \leq U_p(T)$$

where

$$U_p(T) = Q(T) \prod_{i=1}^{m} \rho_i \prod_{j=1}^{n} \alpha_j \mu_{3,j} \tag{2}$$

and

$$\begin{aligned} L_p(T) \;=\; & Q(T-\Delta) \prod_{i=1}^{m} \rho_i \left[1 - \epsilon_i \mu_{2,i} - \frac{\mu_{2,i}^2 + \sigma_{2,i}^2 + \rho_i^2}{r_i^2} \right] \\ & \times \prod_{j=1}^{n} \alpha_j \left[\mu_{3,j} - \frac{(\alpha_{3,j} + \beta_{3,j})(\mu_{3,j}^2 + \sigma_{3,j}^2)}{2} - \frac{\mu_{3,j}^2 + \sigma_{3,j}^2}{s_j} \right] \end{aligned} \tag{3}$$

Computation of the α, μ, σ, and ρ values is standard. The following suggestions for r_i, s_i, and bounds on $Q(T)$ are given in [3]:

$$r_i = \left(2T(\mu_{2,i}^2 + \sigma_{2,i}^2)\right)^{1/3}$$

$$s_j = \left(\frac{T(\mu_{3,j}^2 + \sigma_{3,j}^2)}{\mu_{3,j}}\right)^{1/2}$$

$$\frac{\prod_{i=1}^{k}(\lambda_i T)}{k!}\left(1 - \frac{T}{k+1}\sum_{i=1}^{k}(\lambda_i + \gamma_i)\right) \leq Q(T) \leq \frac{\prod_{i=1}^{k}(\lambda_i T)}{k!}. \quad (4)$$

An important characteristic of these bounds is that they depend only on a small amount of information pertaining to the path. In fact, the products in Equations (2)–(4) can be accumulated in a small, fixed amount of storage space as a path is extended. For computational reasons (for $Q(T)$) we do separately save the λ_i and γ_i values from each class 1 transition, but this requires the storage of only two floating point numbers per transition.

One way to use these bounds is to explore all paths from I to $\mathcal{D}$. Whenever a path $p \in \mathcal{P}$ is discovered, $L_p(T)$ and $U_p(T)$ are computed and added to accumulating totals $L(T)$ and $U(T)$. It is important to prune loops, or other paths with very small (relative) probabilities. SURE-based tools typically prune a path p once $U_p(T)$ is smaller than some threshold ϕ (which may be given by the user, or can be found automatically). Upon pruning p, $U_p(T)$ is added to an accumulating total $P(T)$; the final lower and upper bounds on system failure by time T are then $L(T)$ and $U(T) + P(T)$. One typically desires to find ϕ such that $P(T)$ is an order of magnitude smaller than $U(T)$.

A user of the original ASSIST/SURE toolset constructs a state-space using ASSIST, and analyzes it using SURE. In the next section we describe how the generation and analysis can be combined, and how the whole process is easily parallelized.

5 Analysis Techniques

This section describes ASSURE's technique of depth-first generation and analysis of a model, parallelization of this method, and a user-assisted technique for trimming the model during its generation and analysis. We demonstrate empirically that these techniques effectively accelerate the solution time of some large ASSURE models.

5.1 Depth-First Generation and Analysis

Memory usage seriously degrades the execution time of ASSIST and SURE on very large state-spaces. Not only may tens of megabytes be required to store the model, but both the generation and analysis processes may suffer thrashing in a virtual memory system.

We can address the problem by trading off computational efficiency for space efficiency. ASSIST stores all generated states; upon creating a state it looks to see if that state already exists, and extends a path through that state only upon its initial discovery. A different approach is to simultaneously generate and analyze the state-space along a path, and to discard discovered states once they are no longer needed *for that path.* This provides a significant memory savings since memory requirements are proportional only to path length times fanout. The price paid for memory efficiency is the recomputation of state descriptions. This tradeoff works to our advantage for an important class of problems. As we will see, on the large examples we have studied the benefits of memory efficiency are evident. Furthermore, the approach lends itself to parallel processing (which was our initial consideration) because distinct paths can be generated and analyzed separately on different processors. However, the approach has its limitations. Best results are obtained when the system of interest tolerates only a few number of failures within the mission time, say, 5 or fewer. Beyond that, the combinatorics of the approach threatens to create unacceptable solution times.

Our tool ASSURE combines the functions of ASSIST and SURE as follows. A path p is represented internally by a data structure we call a *path-record.* A path-record contains a copy of every ASSIST state variable, whose values represent the last state on the path. A path-record also contains a list of the λ_i and γ_i values of all class 1 transitions on the path, and accumulated products for Equations (2)–(4). ASSURE begins by initializing a path-record to reflect I, and places it on a *working list.* ASSURE enters a loop where the first path-record on the working list is removed and $U_p(T)$ is computed and compared against the pruning threshold. If $U_p(T)$ is sufficiently high, the path-record's state variables are checked against all death-state conditions. The code that performs this check is C code translated from ASSIST DEATHIF statements. A path-record that survives pruning and death-state testing is subjected to extension through all possible transitions, by checking its state variables against every TRANTO condition specified in the ASSIST model. Every time a TRANTO condition evaluates to **true** a copy of the path-record is created, its state variables are modified as proscribed by the ASSIST model, and the value of the transition rate specified following the transition's BY keyword is recorded. Again, these tests and modifications are performed by C code translations of ASSIST model statements. By testing the path-record against all TRANTO conditions we discover and generate all transitions possible from the path-record's last state. Given these transitions and their rates, all the quantities needed by the SURE bounds for each new path are computed, and recorded in each new path-record. The new path-records are attached to the head of the working list, and the process continues until the working list is empty.

The description above shows that ASSURE generates all sufficiently probable paths from I to $\mathcal{D}$ via a depth-first generation and analysis strategy. In addition, ASSURE provides the additional capability of determining whether a model can survive any K failures without entering a death-state. This is easily encorporated by recording the number of slow transitions on the path, and prune once that count reaches K. If no death-states are uncovered, then the system model survives

any combination of K failures.

It is important to observe that the techniques described above do not depend on the specifics of the ASSIST language. Any formal description of a reliability model will do, provided that one can automatically and quickly find all transitions and their rates from any given state system state. Indeed, as a follow-on to ASSURE, we have built an object-oriented language and tool, REST, that is based on these same principles [15]. Within that framework we have also written a SAVE-to-REST translator, thereby providing transient SAVE models with the computational advantages described in this paper. Furthermore, we believe other tools could also incorporate such an approach. For instance, HARP is widely used, but encounters memory problems on large models [18]. Since HARP analysis is based on a Markov chain, and since system death conditions are recognizable from the defining fault-tree, one could apply a depth-first combined state-space generation and analysis method as we have done with ASSURE. Upon reaching a death-state or a pruned state one could examine the path and numerically compute the exact probability (by uniformization [16]) of reaching that state by time T.

At any time, the memory requirements of ASSURE are basically those of storing the working list. However, ASSURE ends up doing more computation to generate the state-space than does ASSIST. The tradeoff often works to ASSURE's advantage. On moderately large ASSIST models (that lack complex reconfiguration), ASSURE runs ten to twenty times faster than does ASSIST/SURE. A simple analysis helps to quantify the tradeoff. Component failures essentially drive changes in a system state. Let X be a state-vector with N components, and suppose that any given collection of failures results in the same state of X regardless of the sequence in which the failures occur. Ignoring effects of possible aggregation (i.e., different collections of failures resulting in the same state), a state s defined by j failures will lie on $j!$ different paths. But s has j immediate predecessors, implying that ASSIST will discover s exactly j times. To a first approximation then, if the model tends to tolerate j failures before entering a death-state or being pruned, ASSURE does $j!/j = (j-1)!$ times more computation than does ASSIST. On the other hand, ASSURE's memory requirements are small enough that it tends to operate without page faults, whereas ASSIST is observed to thrash on large models. We estimate that ASSIST's average cost of "touching" a state is several hundred times higher than ASSURE's. These estimates suggest that ASSURE is more efficient than ASSIST when the system model tolerates a handful of errors within the mission time, say, 5 or fewer.

5.2 Parallelization

ASSURE's generation and analysis technique is highly suitable for parallel processing, because processors can independently generate and analyze distinct paths from I to $\mathcal{D}$. One way is have a controller process generate all one-step paths $p_1, p_2, \ldots, p_N$ from the initial state I, then distribute these path-records among processors to seed their working lists. Once seeded, a processor is free

to execute exactly as in the serial case. Each processor then independently accumulates a pruning bound, and SURE bounds. The overall bounds are obtained by adding the contribution from each processor.

The method above has the disadvantage that one processor may complete its work long before another processor does. We have used two different methods of dealing with this problem. ASSURE has been parallelized on a 32-processor Intel iPSC/860 multiprocessor. This machine has a fast communication network, making it feasible for a processor with excess path-records to send some to deficient processors. We implemented and studied several dynamic load-balancing schemes that balance the number of path-records per processor nearly perfectly when called. Our studies found that for ASSURE problems it didn't matter very much *how* the load was rebalanced, so long as it was rebalanced. A discussion of the different schemes studied is given in [14].

This type of balancing is not suitable for a loosely coupled network of workstations. However, load-balancing here is simple if the workstations share a common file system. The approach is to have every workstation generate path-records for every descendent of the initial state, and enumerate them the same way. Every workstation i begins by seeding its working list with descendent i; some dedicated process writes the number of unassigned descendents into a commonly viewed file. A workstation executes independently until its working list is empty, at which point it consults the remaining descendent count file to look for more work. If the value in the file is non-zero, the workstation decrements the count, and reseeds its working list with the appropriate descendent. Otherwise, the workstation writes its own results into a reserved file. The computation is complete once all workstations have attempted and failed to acquire additional workload. A monitoring process accumulates and reports the individual workstation results. ASSURE does all this automatically, given a run-time option specifying the network names of machines to use.

The simplicity and power of parallelizing SURE bounds calculations gives us reason to believe that extremely large models can be generated and analyzed in a reasonable amount of time. For example, we considered two different ASSURE models of example flight-control network. The first, called NetA, has 61 elements in its state-vector. Recovery from network failure is simplified enough to be expressed in pure ASSIST. The second model, NetB, uses our language extensions, and has 83 elements in its state-vector. Each hardware component (channel, network interface, link, node, device interface, device) has its own failure rate; the ratio of the fastest recovery rate to the slowest failure rate is 10^{10}. In the data reported below, a path-record was pruned as soon as the upper bound on its probability dropped below $\phi = 1e\text{-}15$. The analysis of NetA generated 3.6 million nodes with an average number of failures when a path terminated of 3.9. If we estimate the actual size of the state-space explored as W choose L (i.e., $W!/((W-L)!L!)$) where W is the length of the state vector and L is the number of component failures, then the NetA analysis is of approximately 0.5 million unique states. NetB generates 57.5 million nodes, with 4.0 average failures at termination, and an estimated true state-space size of 2 million states. We report experiments conducted on 32 processors of the Intel iPSC/860 multiprocessor (based on the i860

Model	1 Sparc	6 Sparcs	12 Sparcs	18 Sparcs	1 i860	32 i860s
NetA (3.6M nodes)	22 min	4 min	2 min	2 min	2.66 min	0.15 min
NetB (57.5M nodes)	7.5 hr	75 min	35 min	24 min	79.5 min	2.5 min

Table 1: Timings of first model problem on parallel platforms

CPU), and on a local area network using 6, 12, and 18 SUN Sparc workstations of various models. The iPSC/860 was dedicated to the application, whereas the network runs were competing with everything else on the network (which was lightly loaded) at the time. Also, our network timings have a resolution of only one minute, being taken from last-modification times on files. Table 1 presents these results.

The primary conclusion we draw from these timings is that the parallelization techniques work to dramatically reduce solution time. Furthermore, while the performance shown is nearly an order of magnitude faster on a dedicated multiprocessor, one can still get impressive performance from workstation networks commonly found in research labs.

5.3 Model Trimming

The combinatorial growth of models explored by our method encourages us to search for ways of reducing the number of states generated and analyzed. For instance, consider a path that has undergone j failures, and suppose we could bound the probability of entering a death-state following any two additional failures. Instead of generating the model for an additional two levels we might trim it, and accumulate the death-state probability bound in the pruning sum. We have developed a way of doing exactly that, and observe significant reductions in the model size. The method is a generalization of the notion of a *trimming bound*, described in [19].

Consider all slow transitions out of an arbitrary state X. Typically each one is related to the failure of a component or to a set of components. Some transitions may lead immediately to death-states; call these transitions *unsafe*. At the other extreme, there are transitions which are inherently safe. Formally, we'll say that a transition δ from state X is *safe* if taking δ from X does not lead to system failure, and if X' is any state reachable from X by one transition then δ may be taken from X' without leading to system failure. For example, a safe transition is defined if a system has a 4-plex redundant subsystem with all components up, and the subsystem remains up so long as 3 components are up. At any time, from any state, the 4-plex can lose its first component safely. Finally, we call a transition *conditionally safe* if the system does not enter a death state by taking it, nor will a death-state be entered if a safe transition is taken first.

Our method is different from [19] in that we make a distinction between safe and conditionally safe transitions. Like the earlier work, we assume (i) that components fail at a low constant rate,

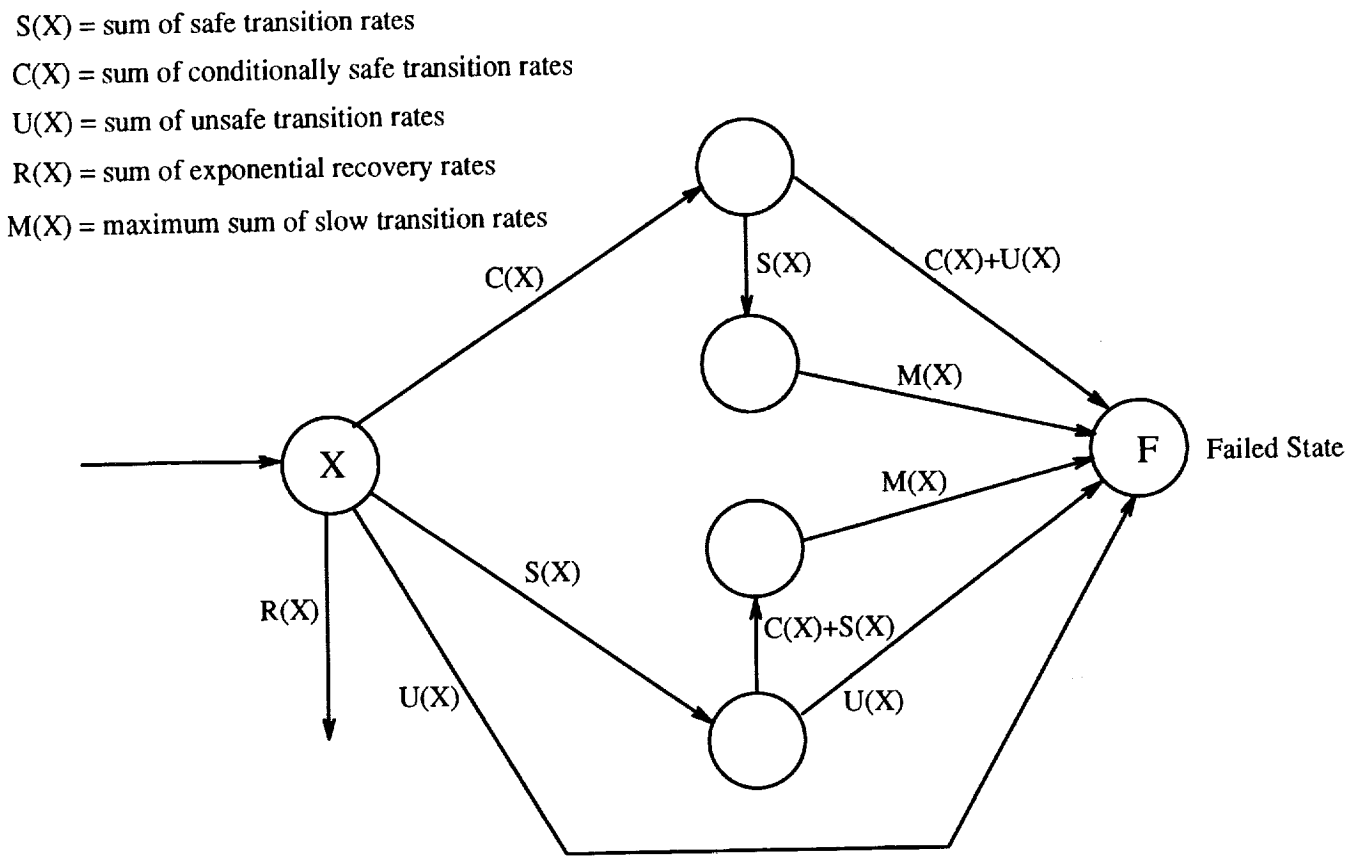

Figure 2: Markov chain used to construct the trimming bound

(ii) fault recovery depends only on the time since fault occurrence, and (iii) all transitions to system failure are component failure transitions.

Now from any state X, let U(X), C(X) and S(X) be the sum of rates of unsafe, conditionally safe, and safe transitions, respectively. Also, let R(X) be the sum of rates of exponentially distributed recovery transitions from X, and let M(X) be an upper bound on the sum of slow transition rates in any state reachable from X in two transitions. To construct a trimming bound we may consider the behavior of a simple Markov chain shown in Figure 2. From X it describes an aggregate recovery, an aggregate unsafe transition, aggregate conditionally safe transition, and aggregate safe transition. The chain also expresses a second level of behavior, with unsafe and not-unsafe transitions. The rates on the second level transitions are upper bounds on the aggregate rates in the actual system. The effect of recovery transitions on these states are omitted, which serves to accelerate the simple chain towards failure state F even faster than the actual system. Our trimming bound is given by adding the SURE upper bounds on each of five paths which extend the path to X further to the failed state F. This sum is greater than the sum of probabilities of reaching any death state eventually reachable by taking a failure transition from X.

In theory one could use the trimming bound by comparing it to a threshold ϕ (like the pruning threshold). If ϕ is larger, the only transitions from X that are generated are the recoveries, and the trimming bound is added to the accumulating pruning bound. In practice it is difficult for a general tool such as ASSURE to automatically compute the necessary failure rates (note, however, that this is less of a problem with tools that impose more structure on their model input description, from which the rates might be inferred). We've addressed the problem in ASSURE by allowing a user to write a C language function that computes $U(X)$, $C(X)$, $S(X)$, $R(X)$, and $M(X)$ for any state X. ASSURE then automatically invokes and uses the results of the routine. This mechanism allows a modeler to exploit knowledge of the system structure in order to quickly compute these transition rates, or upper bounds upon them (or a lower bound on $R(X)$).

Consider our first example problem. When a device in a quad fails it is considered to be "failed" and "in use" until completion of a recovery transition that takes it off-line. A "bad" component is one that is in this transitional state. The ASSURE model defines the system to fail if any of the following conditions holds.

- A fault occurs in one network partition while the other partition is under repair.
- The number of FTP channels that are good is zero, or is less than or equal to the number of channels that are bad.
- For every device type, the number of devices that are good is zero, or less than or equal to the number of devices that are bad.
- A failed network is unable to establish a virtual bus to operative devices.

Using this information, one can write a routine that examines the model state variables and classifies the effect of every component failure as being safe, conditionally safe, or unsafe. Since the classification will be done often it is important to do it quickly. One can always misassign a transition to a class with less safety, e.g., assign what is actually a safe transition to the conditionally safe class. The bounds needed by ASSURE are obtained by summing rates within a class. For example, suppose X reflects a state in our model NetB where one partition is under repair. Then every transition related to a component failure that might trigger a network recovery in the *other* partition is classified as unsafe, e.g., the failure of a link on the virtual bus. On the other hand, transitions related to FTP channel failures may be in any of the three classes. If there is only one good FTP its failure will cause system failure, and hence that transition is unsafe. If there are two good FTP channels, and two failed channels then each FTP channel failure transition is conditionally safe; with three good FTP channels and one failed channel each transition is safe. Other component failures may be similarly analyzed.

A simple modification of this scheme deserves special comment. While ASSURE needs user assistance to produce U(X), C(X), S(X), and R(X), it does not need help computing M(X), provided

that $M(X_i)$ is no larger than the sum of slow transition rates out of the initial state. For this reason ASSURE provides an automatic trimming bound from state X where it is assumed that all slow transitions from X are unsafe, and have a aggregate failure rate bounded by that of the initial state. (HARP trimming makes a similar assumption.) When initiating any solution run, ASSURE can be told not to assume boundedness.

To investigate the utility of these trimming options we solved the first model problem (NetB) using three different options, all with a trimming (or pruning) threshold of ϕ = 1e–15. The first option we call *standard*—a path is pruned if the upper bound for that path is less than ϕ. We call the second option *bounded*—this method uses the automatic bounded trimming method, and needs no user assistance. The third option we call *user-assisted*, because the user supplies a routine that computes and classifies transition rates. The table below illustrates the results, noting the total number of states generated, the pruning bound, and the time required for solution on a single Sparc 1+ workstation. All methods obtained the same unreliability bounds for a 3 hour mission, 3.77e–9 and 3.96e–9.

Method	Total Number of States	Pruning Bound	Execution Time
Standard	57.5M	9.3e–11	590 min
Bounded	7.2M	4.3e–11	112 min.
User Assisted	1.1M	4.5e–11	17 min.

From this data we see the tremendous advantage of exploiting a monotonic property over not exploiting it, and the further advantage of providing user assistance. It is also interesting to note that the standard method's node execution *rate* is nearly twice as fast as the others, since it suffers no overhead to compute lookahead trimming bounds. However, the overhead of computing more advanced trimming bounds is clearly worth the effort.

The key ingredient to making the user-assisted bounds work well is that the user-supplied routine be able to quickly compute upper bounds on the transition rates. The alternative is to let ASSURE discover these rates (at least U(X) and C(X)+S(X)) by generating the descendents of X. We tested the alternative, and found no performance gains. There is, of course, some danger that a user may misclassify transitions, leading to premature trimming and failure to discover important death-states. However, we believe that trimming of this type may be safely used if the system of interest has structure that permits automatic classification of pending transitions, or if there is a higher level system description language (like TOTAL) where sufficient structure is expressed so that a correct user-assisted pruning routine may be generated automatically.

6 Simulation

Despite the promise of analyzing large state-spaces via parallel processing and smart trimming, the problem remains that gargantuan state-spaces defeat any approach based on exhaustive analysis.

This is especially true in systems which tolerate many failures. Even if a tool can analyze a model, albeit slowly, a modeler may desire loose upper bounds on reliability in the course of exploring a model design. Alternatively, one may first wish to exhaustively test to ensure that any combination of K failures will not cause system failure, and then get a rough estimate of reliability. In such cases a Monte Carlo simulation approach can help. This section outlines such an approach, based on importance sampling. We first discuss the mathematics of sampling and show that the basic method is sound. We also point out that importance sampling based on SURE bounds achieves variance reduction over another standard method. We then consider parallelization, and observe excellent speedups. Next we discuss optimized death-state checking, and also further language extensions to support general statistical measurements. Finally we discuss some important implementation considerations.

6.1 Mathematical Basis

For any path p ending in $\mathcal{D}$ (i.e., $p \in \mathcal{P}$), let $f(p)$ denote the probability that the system chooses p on its way to $\mathcal{D}$, if left to run sufficiently long. Then

$$\Pr\{\text{System failure by time } T | \text{path } p \text{ is taken } \} = \Pr\{S_p \leq T | \text{path } p \text{ is taken}\}.$$

Thus

$$\begin{aligned} \Pr\{\text{System failure by time } T\} &= \sum_{p \in \mathcal{P}} f(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= E_f[\Pr\{S_P \leq T | \text{path } P \text{ is taken}\}] \end{aligned} \tag{5}$$

where P is the random path chosen to $\mathcal{D}$. A Monte Carlo approach is to estimate this expectation via random sampling of $\Pr\{S_P \leq T | P \text{ is taken}\}$.

Given a path p and SURE bounds $L_p(T)$ and $U_p(T)$, we know that

$$\frac{L_p(T)}{f(p)} \leq \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \leq \frac{U_p(T)}{f(p)}. \tag{6}$$

This inequality could be used to estimate bounds on $E_f[\Pr\{S_P \leq T | \text{path } P \text{ is taken}\}]$, but there is a serious problem with such an approach. When P is sampled from f, from any state with both fast and slow transitions we will almost always chose the fast (recovery) transition. The majority of death-states occur in those rare cases when recovery mechanisms are defeated by low probability additional failures. Sampling paths using f means missing some of the death-states one is attempting to find. This problem has been recognized before [13, 4, 10, 8], where the notion of *importance sampling* is used. Intuitively, importance sampling is used to skew the path sampling towards rare events. Mathematically, let $g(p)$ be a different probability mass function for sampling

paths such that $g(p) \neq 0$ whenever $f(p) \neq 0$. Then

$$\begin{aligned} E_f[\Pr\{S_P \leq T | P \text{ is taken}\}] &= \sum_{p \in \mathcal{P}} f(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= \sum_{p \in \mathcal{P}} \frac{f(p)}{g(p)} g(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= E_g[R(P) \Pr\{S_P \leq T | \text{path } P \text{ is taken}\}] \end{aligned}$$

where $R(p) = f(p)/g(p)$.

To use importance sampling is to estimate the latter expectation by randomly sampling (with respect to g) bounds on $R(p) \Pr\{S_p \leq T | p \text{ is taken}\}$. From inequality (6) we see that for any path p

$$R(p)\frac{L_p(T)}{f(p)} \leq R(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \leq R(p)\frac{U_p(T)}{f(p)},$$

or equivalently,

$$\frac{L_p(T)}{g(p)} \leq R(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \leq \frac{U_p(T)}{g(p)}. \tag{7}$$

The Monte Carlo analysis consists of sampling (with respect to g) many independent replications of paths to $\mathcal{D}$, and for each computing $L_p(T)/g(p)$ and $U_p(T)/g(p)$ as samples. Following many replications we compute confidence intervals on $E_g[R(P)L_P(T)]$ and $E_g[R(P)U_P(T)]$, and use these to construct confidence intervals on the probability of system failure by T.

Many different ideas have been suggested for important sampling, e.g., see [10]. We have been successful with a strategy that partitions transitions from a state into slow and fast classes, chooses the slow class with some probability q and chooses the fast class with complimentary probability. Within a class a transition is chosen with probability proportional to its transition rate. For a given path p, $g(p)$ is computed as the product of the probabilities of each forced transition decision. This transition selection strategy was proposed in [13]. However, part of that proposal is to also sample holding times, conditioning them on no transition time exceeding T. When $\mathcal{D}$ is reached, the sample statistic is of the form $d(p)f(p)/g(p)$, where $d(p)$ is the product of ratios of the form $h(t_i|t_{i-1}, k)/\hat{h}(t_i|t_{i-1}, k)$. Here t_i is the sampled transition time from the i^{th} state on p, say k, given that k is entered at time t_{i-1}. $h(t_i|t_{i-1})$ is the density of that transition time using k's true holding time distribution, and $\hat{h}$ is the forced density function. Contrast the measure $d(p)f(p)/g(p)$ with the SURE-based measure $U_p(T)/g(p)$. A key point is that conditioned on taking path p, $d(p)f(p)/g(p)$ is still a random variable ($d(p)$ varies), whereas $U_p(T)/g(p)$ is deterministic. This immediately implies that the expected average measure in the original scheme has a larger variance does the expected average SURE measure. Therefore confidence intervals based on SURE bounds (when using the same transition selection strategy) will on average be smaller.

Model	Size (nodes)	Exact Bounds	Solution time	Simulation Bounds (10,000 replications)	Solution time
aclust3	34623	4.94e-8, 4.95e-8	9 sec	(5.00 ± 0.14)e-8, (5.01 ± 0.14)e-8	31 sec
arcsxod1	1032	6.81e-3, 6.92e-3	0.4 sec	(6.63 ± 0.25)e-3, (6.74 ± 0.26)e-3	21 sec
billee	991	2.24e-7, 2.25e-7	0.4 sec	(2.25 ± 0.11)e-7, (2.25 ± 0.11)e-7	27 sec
NetA	3.6M	4.02e-6, 4.12e-6	22 min	(3.47 ± 0.51)e-6, (3.82 ± 0.57)e-6	2.2 min
NetB	57.5M	3.76e-9, 3.97e-9	7.5 hr	(3.68 ± 0.28)e-9, (4.04 ± 0.35)e-9	5.4 min

Table 2: Comparison of SURE-based and simulation-based analysis

The quality of results obtained from importance sampling schemes are known to be sensitive to the problem class. We were naturally concerned whether the schemes we examined were effective on problems for which SURE was intended. Happily, the scheme above with $q = 0.5$ has proven to give results consistent with SURE analysis (this setting was also recommended in [4]). We tested the simulation-based results with SURE predictions, on a suite of problems used at NASA to validate ASSIST. Three of these are listed below, as well as models NetA and NetB, described earlier, using standard pruning. All simulation runs are based on 10,000 replications. The simulation-based lower and upper bounds are given as 95% confidence intervals, and timings are taken on a SUN Sparc workstation. This data suggests that the simulation based approach is able to find small intervals around the exact bounds, and in the case of the very large models do so more rapidly than the exact analysis. However, it is also clear that orders of magnitude more replications are needed if we wished to shrink the confidence intervals to less than one percent of the mean. The advantage of simulation is that reasonably good numbers can be gotten relatively quickly. We expect there is utility in numbers known to be uncertain within 10%.

We also estimated reliability on the models above using skewed holding times as described in [13, 8]. On the small models there was no appreciable difference between the relative errors (confidence interval width divided by sample mean) of the two approaches. However, on NetA and NetB the SURE-based approached yielded relative errors that are 20% smaller. The SURE approach also runs 10–20% faster, since it avoids random number generation for holding times. For the 10,000 replications examined here, the confidence intervals for both approaches are not small enough to distinguish between an estimate based on SURE's upper bound, or the estimate of the precise probability.

The primary motivation for importance sampling is variance reduction. It is therefore instructive to examine how the sample variance achieved under our scheme changes as the class probability threshold q changes. This is illustrated in Figure 3, where for the NetB model we plot 95% confidence intervals on the upper bound 3.97e-9, following 10,000 replications. This data shows

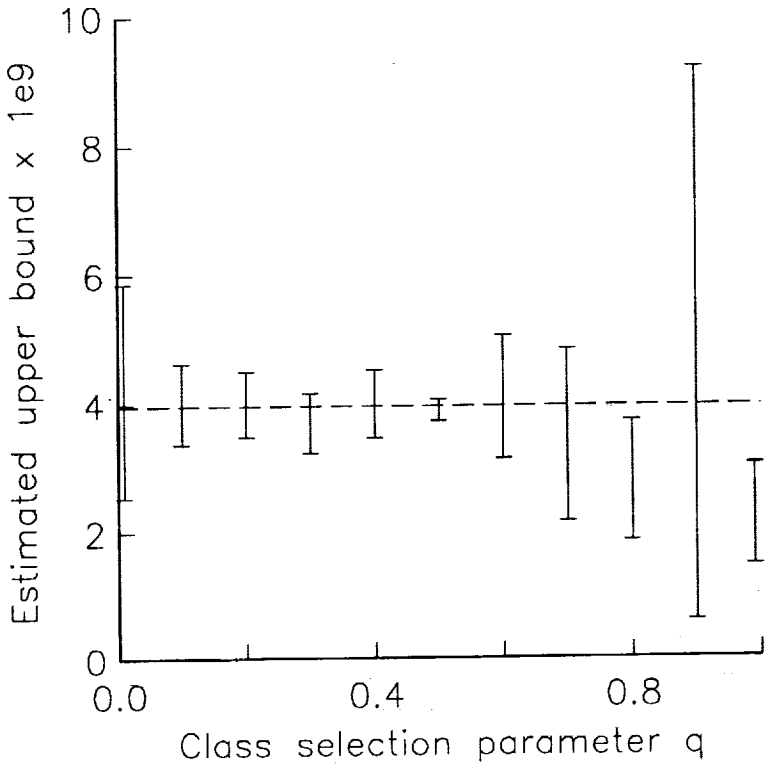

Figure 3: Confidence intervals as function of importance sampling parameter q

the danger of skewing q too far one way or the other, $q = 0.5$ appears to be a satisfactory setting. However, since effective importance sampling is known to be problem class dependent, ASSURE can call a user written routine to do the importance sampling. Such a routine is passed a description of the system state, and all transitions possible from that state (and their rates). The routine chooses a transition, and reports back the probability of making that choice under the importance sampling strategy. This is all the information ASSURE needs to correctly compute its statistics.

6.2 Parallelization

Simulation replications are trivially parallelized; we have done so on the workstation network. The only challenge is to use a load-balancing scheme that does not incur excessive overhead, but which is responsive to changing network loads. Our scheme is to maintain a commonly accessed file, to contain the remaining number of replications. A workstation devoid of work accesses this file, acquires some fixed number G replications (G is user-defined so that replications are acquired no more than, say once a minute), modifies the file and releases it. The simulation is complete after every workstation finishes its work, and sees zero remaining replications. A monitoring process combines and reports the aggregate results. To demonstrate the effectiveness of this approach, we simulated NetB for 100,000 replications on 1, 6, 12, and 18 workstations at a time when the

network load was low. The running times were respectively 35, 6, 3, 2 minutes. Once again we see the tremendous advantage offered by parallel processing.

6.3 Additional Issues

We now consider some auxiliary issues. In ASSURE, simulation-based analysis generates different problem-dependent code than does exact analysis; the generation of a state's descendents is done in two passes. The first pass identifies the *existence* of each descendent, and its transition rate. This pass does not actually perform the state-space modification. ASSURE's simulation control code selects a descendent, and in a second pass that descendent's state is created. We judged this approach to be crucial for problems with large state vectors, and/or complex state-modification routines. Indeed, this approach yielded a factor of two reduction in execution time on NetA and NetB.

Our study of the second model problem led us to consider another implementation issue, that of death-state checking. ASSURE's basic scheme checks death-state conditions after every transition. This makes sense for many models, including all of the ones we've considered so far in this paper. However, consider our second model problem, where a system is considered to have entered $\mathcal{D}$ if there exist two operable processors that cannot communicate under the constraints of the fault-tolerant routing protocol. We noted earlier the high computational cost of checking that condition. The problem is that a path may be extended many times before reaching $\mathcal{D}$; most of the death-state checks are unnecessary, as they do not observe a death-state.

We exploit the fact that once the system state enters $\mathcal{D}$ it will not depart. The optimization is to only periodically check whether a path under expansion has entered $\mathcal{D}$, say, check every d transitions. We keep an ordered list of all path-records generated in the last d transitions. Upon reaching the d^{th} transition since the last check, we check the DEATHIF conditions on the present state variable values. If the state is not in $\mathcal{D}$ we release the first $d-1$ of the stored path records, and continue for another d transitions. Once a death-state is uncovered we must find the *first* state to enter $\mathcal{D}$ among the last d visited. Since their path-records have been saved in order of generation, we may perform a binary search. The number of death-state checks is thus approximately logarithmic in the path length, rather than linear. Observe that when systems can be repaired it may be possible to express a model that can pass into and out of $\mathcal{D}$, even though that may not be intended. For this reason, the optimization under discussion must be requested by a user, it is not automatic. However, one could adapt the scheme by always checking for a death-state on recognition of a recovery transition—we check the state just prior to the recovery and use that state as the terminus of a search interval.

In order to both illustrate the advantage of periodic checking and illustrate that simulation based analysis can handle large problems, we consider the second model problem. The routing protocol studied permits at most two "miss-steps", which means that the number of links crossed

Hypercube Dimension	Size (nodes,links)	Avg. Failures	Constant Check sim. rate	Periodic Check sim. rate
6	(64,192)	73	4.52 reps/min	28.8 reps/min
7	(128,448)	166	0.4 reps/min	5.2 reps/min
8	(256,1024)	400	0.04 reps/min	0.92 reps/min

Table 3: Simulation rates on fault-tolerant routing problem, as the problem size varies

when i and j communicate is no larger then four plus the Hamming distance between i and j. With some straightforward optimizations it costs $O(nodes \times links)$ time to determine whether a given network configuration is dead. We check the death-state condition every 100 transitions. Table 3 shows the effect on the simulation rate (replications/minute) of the "constant check" and "periodic check" methods, as the size of the problem increases. The table shows the problem size (in numbers of components), the average number of failed components when $\mathcal{D}$ is entered, and the simulation rates. This data clearly shows the advantage of periodic checking on problems of this type, and also shows that simulation-based ASSURE analysis is able to deal with relatively large problems, especially if we use parallel processing. On the largest problem shown here, we could expect to complete a 1000 replication run in approximately an hour using 18 workstations in parallel.

On this data the relative error from the SURE-based approach is approximately 33% smaller than that using skewed sample times, showing again the variance-reduction advantage of using SURE bounds.

Needs of the second model problem also gave rise to another language extension. The users were interested in obtaining statistical information about the system configuration in death-states, e.g., the average number of failed nodes and/or links. It was relatively easy to provide this by allowing "statistics" variables to be declared in the ASSIST model, e.g.,

```
SAMPLE FailedNodes;
```

A user provides a routine, called when a death-state is recognized, that assigns values to all `SAMPLE` variables. ASSURE automatically computes averages and confidence intervals, reporting these at the end of the analysis.

7 Conclusions

This paper demonstrates methods for accelerating the solution time of reliability analyses based on the SURE bounds. Our methods are centered around the notion of simultaneously generating and analyzing a state-space along a failure path, but discarding the state information once the path is analyzed. This provides a significant memory savings, but exacts a cost of recomputing state

information. We have shown that this tradeoff is advantageous when system failure occurs after a small number of component failures. In addition the approach is easily parallelized, either on a dedicated multiprocessor or on an ordinary network of workstations. An important part of our method is to use a minimum of specialized syntax to describe a framework for a model's transition behavior, and to let a modeler use the full resources of the C programming language to describe the details of that behavior.

We also investigate the integration of SURE bounds and Monte Carlo simulation based on importance sampling. We find that the approach produces accurate results using as few as 10,000 replications on models with two orders of magnitude more states. Consequently, on large models the simulation-based analysis executes more quickly. Furthermore, we observe that SURE-based Monte Carlo estimation has desirable variance reduction properties. Finally, simulation-based analysis admits solution of problems that are too large for exact analysis, and admits easy exploitation of parallelism by simulating independent replications in parallel.

All of the methods described are encorporated in a tool called ASSURE. From a single user interface, ASSURE provides exact analysis or simulation-based analysis, serial execution or parallel execution. Empirical studies of large models solved with ASSURE show that the methods we describe are effective in accelerating the solution time of large complex problems.

Our results show the promise of attacking large reliability problems by path analysis. Further work may be directed towards generalizing the SURE bounds to include non-homogeneous failure rates, and to sharpen confidence intervals with more advance importance sampling schemes.

Distribution

ASSURE is available by request. Contact the first author at `nicol@cs.wm.edu` .

References

[1] M. Boyd and S. Bavuso. Simulation modeling for long duration spacecraft control systems. In *Proceedings of 1993 Annual Reliability and Maintainability Symposium.* IEEE Press, Jan. 1993.

[2] R. Butler. An abstract language for specifying Markov reliability models. *IEEE Trans. on Reliability*, R-35(5):595–601, December 1986.

[3] R.W. Butler and A.L. White. SURE reliability analysis. NASA Technical Paper 2764, NASA Langley Research Center, March 1989.

[4] A. Conway and A. Goyal. Monte Carlo simulation of computer system availability/reliability models. In *Proceedings of the* 17^{th} *International Symposium on Fault-Tolerant Computing.* CS Press, July 1987.

[5] J.B. Dugan, K. Trivedi, M. Smotherman, and R. Geist. The hybrid automated reliability predictor. *AIAA Journal of Guidance, Control and Dynamics*, 9(3):319–331, May-June 1986.

[6] J.B. Dugan. Fault trees and imperfect coverage. *IEEE Transactions on Reliability*, R-38, June 1989.

[7] A. Goyal et al. The system availability estimator. In *Proceedings of the* 16^{th} *Int'l Symposium on Fault-Tolerant Computing*, pages 84–89. CS Press, 1986.

[8] R. Geist and M. Smotherman. Ultrahigh reliability estimates through simulation. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 350–355. IEEE Reliability Society, January 1989.

[9] K.K. Goswami and R.K. Iyer. DEPEND:A design environment for prediction and evaluation of system dependability. In *Proceedings of the* 9^{th} *Digital Avionics Systems Conference.* IEEE Press, Oct. 1990.

[10] A. Goyal, P. Shahabuddin, P. Heidelberger, V. Nicola, and P. Glynn. A unified framework for simulating Markovian models of highly dependable systems. *IEEE Trans. Computers*, 41(1):36–51, January 1992.

[11] S. Johnson. The ASSIST language user's manual. NASA Technical Memorandum 87735, NASA Langley Research Center, 1986.

[12] J.H. Lala. Advanced information processing system (AIPS) - based fault tolerant avionics architecture for launch vehicles,. In 9^{th} *Digital Avionics Systems Conference*, pages 125–132, October 1990.

[13] E. Lewis and F. Bohm. Monte Carlo simulation of Markov unreliabilty models. *Nuclear Engineering and Design*, 77:49–62, 1984.

[14] D. Nicol. Communication efficient global load balancing. In *Proceedings of the 1992 Scalable High Performance Computing Conference.* IEEE Press, April 1992.

[15] D. Nicol, D. Palumbo, and A. Rikfin. REST:A parallelized system of reliability estimation. In *Proceedings of the Annual Reliability and Maintainability Symposium.* IEEE Reliability Society, 1993.

[16] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

[17] R. Sahner and K. Trivedi. Reliability modeling using SHARPE. *IEEE Transactions on Reliability*, R-36:186–193, June 1987.

[18] T. Sharma and I. Bazovsky. Reliability analysis of large systems by Markov techniques. In *Proceedings of 1993 Annual Reliability and Maintainability Symposium.* IEEE Press, Jan. 1993.

[19] A.L. White and D.L. Palumbo. State reduction for semi-markov reliability models. In *Proceedings of 1990 Annual Reliability and Maintainability Symposium.* IEEE Press, Jan. 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1993	Contractor Report

4. TITLE AND SUBTITLE
RELIABILITY ANALYSIS OF COMPLEX MODELS USING SURE BOUNDS

5. FUNDING NUMBERS
C NAS1-18605
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
David M. Nicol
Daniel L. Palumbo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-14

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191445
ICASE Report No. 92-14

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Submitted to IEEE Transactions on Reliability

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As computer and communications systems become more complex it becomes increasingly more difficult to analyze their hardware reliability, because simple models may fail to adequately capture subtle but important model features. This paper describes a number of ways we have addressed this problem for analyses based upon White's SURE theorem. We point out how reliability analysis based on SURE mathematics can be extracted from a general C language description of the model behavior, how it can attack very large problems by accepting recomputation in order to reduce memory usage, how such analysis can be parallelized both on multiprocessors and on networks of ordinary workstations, and observe excellent performance gains by doing so. We also discuss how the SURE theorem supports efficient Monte Carlo based estimation of reliability, and show the advantages of the method.

14. SUBJECT TERMS	15. NUMBER OF PAGES
reliability, Markov modeling	27
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1993 - 728-064/66013